IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Method And System For Using A Portion Of A Digital Good As A Substitution Box

Inventor(s):
Mariusz H. Jakubowski
Ramarathnam Venkatesan

ATTORNEY'S DOCKET NO. MS1-528US

REFERENCE TO RELATED APPLICATIONS

This is a continuation-in-part of Application No. 09/536,033, filed March 27, 2000, entitled "System and Method for Protecting Digital Goods Using Random and Automatic Code Obfuscation".

TECHNICAL FIELD

This invention relates to systems and methods for protecting digital goods, such as software.

BACKGROUND

Digital goods (e.g., software products, data, content, etc.) are often distributed to consumers via fixed computer readable media, such as a compact disc (CD-ROM), digital versatile disc (DVD), soft magnetic diskette, or hard magnetic disk (e.g., a preloaded hard drive). More recently, more and more content is being delivered in digital form online over private and public networks, such as Intranets and the Internet. Online delivery improves timeliness and convenience for the user, as well as reduces delivery costs for a publisher or developers. Unfortunately, these worthwhile attributes are often outweighed in the minds of the publishers/developers by a corresponding disadvantage that online information delivery makes it relatively easy to obtain pristine digital content and to pirate the content at the expense and harm of the publisher/developer.

One concern of the publisher/developer is the ability to check digital content, after distribution, for alteration. Such checking, is often referred to as SRI (Software Resistance to Interference). The desire to check for such alterations

can vary (e.g., to ensure that the content continues to operate as intended by the publisher/developer, to protect against improper copying, etc.).

The unusual property of content is that the publisher/developer (or reseller) gives or sells the *content* to a client, but continues to restrict *rights* to use the content even after the content is under the sole physical control of the client. For instance, a software developer typically sells a limited license in a software product that permits a user to load and run the software product on one or more machines (depending upon the license terms), as well as make a back up copy. The user is typically not permitted to make unlimited copies or redistribute the software to others.

These scenarios reveal a peculiar arrangement. The user that possesses the digital bits often does not have full rights to their use; instead, the provider retains at least some of the rights. In a very real sense, the legitimate user of a computer can be an adversary of the data or content provider.

One of the uses for SRI is to provide "digital rights management" (or "DRM") protection to prevent unauthorized distribution of, copying and/or illegal operation of, or access to the digital goods. An ideal digital goods distribution system would substantially prevent unauthorized distribution/use of the digital goods. Digital rights management is fast becoming a central requirement if online commerce is to continue its rapid growth. Content providers and the computer industry must quickly address technologies and protocols for ensuring that digital goods are properly handled in accordance with the rights granted by the developer/publisher. If measures are not taken, traditional content providers may be put out of business by widespread theft or, more likely, will refuse altogether to deliver content online.

DETAILED DESCRIPTION

A digital rights management (DRM) distribution architecture produces and distributes digital goods in a fashion that renders the digital goods resistant to many known forms of attacks. The DRM distribution architecture protects digital goods by automatically and randomly manipulating portions of the code using multiple protection techniques. Essentially any type of digital good may be protected using this architecture, including such digital goods as software, audio, video, and other content. For discussion purposes, many of the examples are described in the context of software goods, although most of the techniques described herein are effective for non-software digital goods, such as audio data, video data, and other forms of multimedia data.

DRM Distribution Architecture

Fig. 1 shows a DRM distribution architecture 100 in which digital goods (e.g., software, video, audio, etc.) are transformed into protected digital goods and distributed in their protected form. The architecture 100 has a system 102 that develops or otherwise produces the protected good and distributes the protected good to a client 104 via some form of distribution channel 106. The protected digital goods may be distributed in many different ways. For instance, the protected digital goods may be stored on a computer-readable medium 108 (e.g., CD-ROM, DVD, floppy disk, etc.) and physically distributed in some manner, such as conventional vendor channels or mail. The protected goods may alternatively be downloaded over a network (e.g., the Internet) as streaming content or files 110.

The developer/producer system 102 has a memory 120 to store an original digital good 122, as well as the protected digital good 124 created from the original digital good. The system 102 also has a production server 130 that transforms the original digital good 122 into the protected digital good 124 that is suitable for distribution. The production server 130 has a processing system 132 and implements an obfuscator 134 equipped with a set of multiple protection tools 136(1)-136(N). Generally speaking, the obfuscator 134 automatically parses the original digital good 122 and applies selected protection tools 136(1)-136(N) to various portions of the parsed good in a random manner to produce the protected digital good 124. Applying a mixture of protection techniques in random fashion makes it extremely difficult for pirates to create illicit copies that go undetected as legitimate copies.

The original digital good 122 represents the software product or data as originally produced, without any protection or code modifications. The protected digital good 124 is a unique version of the software product or data after the various protection schemes have been applied. The protected digital good 124 is functionally equivalent to and derived from the original data good 122, but is modified to prevent potential pirates from illegally copying or otherwise distributing the digital goods to others. In addition, some modifications enable the client to determine whether the product has been tampered with.

The developer/producer system 102 is illustrated as a single entity, with memory and processing capabilities, for ease of discussion. In practice, however, the system 102 may be configured as one or more computers that jointly or independently perform the tasks of transforming the original digital good into the protected digital good.

The client 104 has a secure processor 140, memory 142 (e.g., RAM, ROM, Flash, hard disk, CD-ROM, etc.), one or more input devices 144 (e.g., keyboard, joystick, voice recognition, etc.), and one or more output devices 146 (e.g., monitor, speakers, etc.). The client may be implemented as a general purpose computing unit (e.g., desktop PC, laptop, etc.) or as other devices, such as set-top boxes, audio/video appliances, game consoles, and the like.

The client 104 runs an operating system 150, which is stored in memory 142 and executed on the secure processor 140. Operating system 150 represents any of a wide variety of operating systems, such as a multi-tasking, open platform system (e.g., a "Windows"-brand operating system from Microsoft Corporation). The operating system 150 includes an evaluator 152 that evaluates the protected digital goods prior to their utilization to determine whether the protected digital goods have been tampered with or modified in any manner. In particular, the evaluator 152 is configured to analyze the various portions according to the different protection schemes originally used to encode the good to evaluate the authenticity of the digital good.

Some protection schemes involve executing instructions, analyzing data, and performing other tasks in the most secure areas of the operating system 150 and secure processor 140. Accordingly, the evaluator 152 includes code portions that may be executed in these most secure areas of the operating system and secure processor. Although the evaluator 152 is illustrated as being integrated into the operating system 150, it may be implemented separately from the operating system.

In the event that the client detects some tamper activity, the secure processor 140 acting alone, or together with the operating system 150, may decline

to execute the suspect digital code. For instance, the client may determine that the software product is an illicit copy because the evaluations performed by the evaluator 152 are not successful. In this case, the evaluator 152 informs the secure processor 140 and/or the operating system 150 of the suspect code and the secure processor 140 may decline to run that software product.

It is further noted that the operating system 150 may itself be the protected digital good. That is, the operating system 150 may be modified with various protection schemes to produce a product that is difficult to copy and redistribute, or at least makes it easy to detect such copying. In this case, the secure processor 140 may be configured to detect an improper version of the operating system during the boot process (or at other times) and prevent the operating system from fully or partially executing and obtaining control of system resources.

For protected digital goods delivered over a network, the client 104 implements a tamper-resistant software (not shown or implemented as part of the operating system 150) to connect to the server 102 using an SSL (secure sockets layer) or other secure and authenticated connection to purchase, store, and utilize the digital good. The digital good may be encrypted using well-known algorithms (e.g., RSA) and compressed using well-known compression techniques (e.g., ZIP, RLE, AVI, MPEG, ASF, WMA, MP3).

Obfuscating System

Fig. 2 shows the obfuscator 134 implemented by the production server 130 in more detail. The obfuscator 134 is configured to transform an original digital good 122 into a protected digital good 124. The obfuscating process is usually applied just before the digital good is released to manufacture or prior to being

downloaded over a network. The process is intended to produce a digital good that is protected from various forms of attacks and illicit copying activities. The obfuscator 134 may be implemented in software (or firmware), or a combination of hardware and software/firmware.

The obfuscator 134 has an analyzer 200 that analyzes the original digital good 122 and parses it into multiple segments. The analyzer 200 attempts to intelligently segment the digital good along natural boundaries inherent in the product. For instance, for a software product, the analyzer 200 may parse the code according to logical groupings of instructions, such as routines, or sub-routines, or instruction sets. Digital goods such as audio or video products may be parsed according to natural breaks in the data (e.g., between songs or scenes), or at statistically computed or periodic junctures in the data.

In one specific implementation for analyzing software code, the analyzer 200 may be configured as a software flow analysis tool that converts the software program into a corresponding flow graph. The flow graph is partitioned into many clusters of nodes. The segments may then take the form of sets of one or more nodes in the flow graph. For more information on this technique, the reader is directed to co-pending U.S. Patent Application Serial Number 09/525,694, entitled "A Technique for Producing, Through Watermarking, Highly Tamper-Resistant Executable Code and Resulting "Watermarked" Code So Formed", which was filed March 14, 2000, in the names of Ramarathnam Venkatesan and Vijay Vazirani. This Application is assigned to Microsoft Corporation and is hereby incorporated by reference.

The segments may overlap one another. For instance, one segment may contain a set of instructions in a software program and another segment may contain a subset of the instructions, or contain some but not all of the instructions.

The obfuscator 134 also has a target segment selector 202 that randomly applies various forms of protection to the segmented digital good. In the illustrated implementation, the target selector 202 implements a pseudo random generator (PRG) 204 that provides randomness in selecting various segments of the digital good to protect. The target segment selector 202 works together with a tool selector 206, which selects various tools 136 to augment the selected segments for protection purposes. In one implementation, the tool selector 206 may also implement a pseudo random generator (PRG) 208 that provides randomness in choosing the tools 136.

The tools 136 represent different schemes for protecting digital products. Some of the tools 136 are conventional, while others are not. These distinctions will be noted and emphasized throughout the continuing discussion. Fig. 2 shows sixteen different tools or schemes that create a version of a digital good that is difficult to copy and redistribute without detection and that is resistant to many of the known pirate attacks, such as BORE (break once, run everywhere) attacks and disassembly attacks.

The illustrated tools include oblivious checking 136(1), code integrity verification 136(2), acyclic and cyclic code integrity verification 136(3), secret key scattering 136(4), obfuscated function execution 136(5), code as an S-box 136(6), encryption/decryption 136(7), probabilistic checking 136(8), Boolean check obfuscation 136(9), in-lining 136(10), reseeding of PRG with time varying inputs 136(11), anti-disassembly methods 136(12), shadowing of relocatable

addresses 136(13), varying execution paths between runs 136(14), anti-debugging methods 136(15), and time/space separation between tamper detection and response 136(16). The tools 136(1)-136(16) are examples of possible protection techniques that may be implemented by the obfuscator 134. It is noted that more or less than the tools may be implemented, as well as other tools not mentioned or illustrated in Fig. 2. The exemplary tools 136(1)-136(16) are described below in more detail beneath the heading "Exemplary Protection Tools".

The target segment selector 202 and the tool selector 206 work together to apply various protection tools 136 to the original digital good 122 to produce the protected digital good 124. For segments of the digital good selected by the target segment selector 202 (randomly or otherwise), the tool selector 206 chooses various protection tools 136(1)-136(16) to augment the segments. In this manner, the obfuscator automatically applies a mixture of protection techniques in a random manner that makes it extremely difficult for pirates to create usable versions that would not be detectable as illicit copies.

The obfuscator 134 also includes a segment reassembler 210 that reassembles the digital good from the protected and non-protected segments. The reassembler 210 outputs the protected digital good 124 that is ready for mass production and/or distribution.

The obfuscator 134 may further be configured with a quantitative unit 212 that enables a producer/developer to define how much protection should be applied to the digital good. For instance, the producer/developer might request that any protection not increase the runtime of the product. The producer/developer may also elect to set the number of checkpoints (e.g., 500 or 1000) added to the digital good as a result of the protection, or define a maximum

number of lines/bytes of code that are added for protection purposes. The quantitative unit 212 may include a user interface (not shown) that allows the user to enter parameters defining a quantitative amount of protection.

The quantitative unit 212 provides control information to the analyzer 200, target segment selector 202, and tool selector 206 to ensure that these components satisfy the specified quantitative requirements. Suppose, for example, the producer/developer enters a predefined number of checkpoints (e.g., 500). With this parameter, the analyzer 200 ensures that there are a sufficient number of segments (e.g., >500), and the target segment selector 202 and tool selector 206 apply various tools to different segments such that the resulting number of checkpoints approximates 500.

General Operation

Fig. 3 shows the obfuscation process 300 implemented by the obfuscator 134 at the production server 102. The obfuscation process is implemented in software and will be described with additional reference to Figs. 1 and 2.

At block 302, the quantitative unit 212 enables the developer/producer to enter quantitative requirements regarding how much protection should be applied to the digital good. The developer/producer might specify, for example, how many checkpoints are to be added, or how many additional lines of code, or whether runtime can be increased as a result of the added protection.

At block 304, the analyzer/parser 200 analyzes an original digital good and parses it into plural segments. The encoded parts may partially or fully overlap with other encoded parts.

The target segment selector 202 chooses one or more segments (block 306). Selection of the segment may be random with the aid of the pseudo random generator 204. At block 308, the tool selector 206 selects one of the tools 136(1)-136(16) to apply to the selected section. Selection of the tools may also be a randomized process, with the assistance of the pseudo random generator 208.

To illustrate this dual selection process, suppose the segment selector 202 chooses a set of instructions in a software product. The tool selector 206 may then use a tool that codes, manipulates or otherwise modifies the selected segment. The code integrity verification tool 136(2), for example, places labels around the one or more segments to define the target segment. The tool then computes a checksum of the bytes in the target segment and hides the resultant checksum elsewhere in the digital good. The hidden checksum may be used later by tools in the client 104 to determine whether the defined target segment has been tampered with.

Many of the tools 136 place checkpoints in the digital good that, when executed at the client, invoke utilities that analyze the segments for possible tampering. The code verification tool 136(2) is one example of a tool that inserts a checkpoint (i.e., in the form of a function call) in the digital good outside of the target segment. For such tools, the obfuscation process 300 includes an optional block 310 in which the checkpoint is embedded in the digital good, but outside of the target segment. In this manner, the checkpoints for invoking the verification checks are distributed throughout the digital good. In addition, placement of the checkpoints throughout the digital good may be random.

The process of selecting segment(s) and augmenting them using various protection tools is repeated for many more segments, as indicated by block 312.

Once the obfuscator has finished manipulating the segments of the digital code (i.e., the "no" branch from block 312), the reassembler 210 reassembles the protected and non-protected segments into the protected digital good (block 314).

Fig. 4 shows a portion of the protected digital good 124 having segments i, i+1, i+2, i+3, i+4, i+5, and so forth. Some of the segments have been augmented using different protection schemes. For instance, segment i+1 is protected using tool 7. The checkpoint CP_{i+1} for this segment is located in segment i+4. Similarly, segment i+3 is protected using tool 3, and the checkpoint CP_{i+3} for this segment is located in segment i+2. Segment i+4 is protected using tool K, and the checkpoint CP_{i+4} for this segment is located in segment i.

Notice that the segments may overlap one another. In this example, segment i+3 and i+4 partially overlap, thus sharing common data or instructions. Although not illustrated, two or more segments may also completely overlap, wherein one segment is encompassed entirely within another segment. In such situations, a first protection tool is applied to one segment, and then a second protection tool is applied to another segment, which includes data and/or instructions just modified by the first protection tool.

Notice also that not all of the segments are necessarily protected. For instance, segment i+2 is left "unprotected" in the sense that no tool is applied to the data or instructions in that segment.

Fig. 5 shows the protected digital good 124 as shipped to the client, and illustrates control flow through the good as the client-side evaluator 152 evaluates the good 124 for any sign of tampering. The protected digital good 124 has multiple checkpoints 500(1), 500(2),..., 500(N) randomly spread throughout the good. When executing the digital good 124, the evaluator 152 passes through the

various checkpoints 500(1)-500(N) to determine whether the checks are valid, thereby verifying the authenticity of the protected digital good.

If any checkpoint fails, the client is alerted that the digital good may not be authentic. In this case, the client may refuse to execute the digital good or disable portions of the good in such a manner that renders it relatively useless to the user.

Exemplary Protection Tools

The obfuscator 134 illustrated in Fig. 2 shows sixteen protection tools 136(1)-136(16) that may be used to protect the digital good in some manner. The tools are typically invoked after the parser 200 has parsed the digital good into multiple segments. Selected tools are applied to selected segments so that when the segment good is reassembled, the resulting protected digital good is a composite of variously protected segments that are extremely difficult to attack. The sixteen exemplary tools are described below in greater detail.

Oblivious Checking

One tool for making a digital good more difficult to attack is referred to as "oblivious checking". This tool performs checksums on bytes of the digital product without actually reading the bytes.

More specifically, the oblivious checking tool is designed so that, given a function f, the tool computes a checksum S(f) such that:

(1) If f is not changed, S(f) can be verified to be correct.

(2) If f is changed to f', $S(f') \neq S(f)$ with extremely high probability.

Fig. 6 illustrates an exemplary implementation of an oblivious checking process 600 implemented by the oblivious checking tool 136(1) in the obfuscator 134. The first few blocks 602-606 are directed toward instrumenting the code for function f. At block 602, the tool identifies instructions in the software code that possibly modify registers or flags. These instructions are called "key instructions". Alternatively, other instructions (or groups of instructions) could be the key instructions.

For each key instruction, the tool inserts an extra instruction that modifies a register R in a deterministic fashion based on the key instruction (block 604). This extra instruction is placed anywhere in the code, but with the requirement that it is always executed if the corresponding key instruction is executed, and moreover, is always executed after the key instruction. The control flow of function f is maintained as originally designed, and does not change. Thus, after instrumenting the code, each valid computation path of function f is expected to have instructions modifying R in various ways.

At block 606, the tool derives an input set "I" containing inputs x to the function f, which can be denoted by $I = \{x_1, x_2, x_3 \ldots x_n\}$. The input set "I" may be derived as a set of input patterns to function f that ensures that most or all of the valid computation paths are taken. Such input patterns may be obtained from profile data that provides information about typical runs of the entire program. The input set "I" may be exponential in the number of branches in the function, but should not be too large a number.

At block 608, the tool computes S(f) through the use of a mapping function g, which maps the contents of register R to a random element of I with uniform probability. Let f(x) denote the value of register R, starting with 0, after executing

f on input x. The function f(x) may be configured to be sensitive to key features of the function so that if a computation path were executed during checksum computation, then any significant change in it would be reflected in f(x) with high probability.

One implementation of computing checksum S(f) is as follows:

```
Start with x = x0
Cks := f(x0) XOR x0
For i=1 to K do
     xi  := g(f(xi-1))
     Cks += f(xi) XOR xi.
End for
```

The resulting checksum S(f) is the initial value x_0, along with the value Cks, or (x_0, Cks). Notice that the output of one iteration is used to compute the input of the next iteration. This loop makes the checksum shorter, since there is only one initial input instead of a set of K independent inputs (i.e., only the input x_0 rather than the entire set of K inputs), although all of the K inputs need to be made otherwise available to the evaluator verifying the checksum.

Each iteration of the loop traverses some computation path of the function f. A random factor may optionally be included in determining which computation path of the function f to traverse. Preferably, each computation path of function f has the same probability of being examined during one iteration. For K iterations, the probability of a particular path being examined is:

$$1 - (1 - 1/n)K \approx K/n, \text{ where } n = \text{card } (I).$$

It should be noted that, although various randomness may be included in the oblivious checking as mentioned above, such randomness should be implemented in a manner that can be duplicated during verification (e.g., to allow the checksum S(f) to be re-calculated and verified). For example, the randomness introduced by the oblivious checking tool 136(1) may be based on a random number seed and pseudo-random number generator that is also known to evaluator 152 of Fig. 1.

Code Integrity Verification

Another tool for embedding some protection into a digital good is known as "code integrity verification". This tool defines one or more segments of the digital good with "begin" and "end" labels. Each pair of labels is assigned an identification tag. The tool computes a checksum of the data bytes located between the begin and end labels and then hides the checksum somewhere in the digital good.

Fig. 7 shows a portion of a digital good 700 having two segments S1 and S2. In the illustration, the two segments partially overlap, although other segments encoded using this tool may not overlap at all. The first segment S1 is identified by begin and end labels assigned with an identification tag ID1, or Begin(ID1) and End(ID1). The second segment S2 is identified by begin and end labels assigned with an identification tag ID2, or Begin(ID2) and End(ID2).

The code integrity verification tool computes a checksum of the data bytes between respective pairs of begin/end labels and stores the checksum in the digital good. In this example, the checksums CS1 and CS2 are stored in locations that are separate from the checkpoints.

The tool inserts a checkpoint somewhere in the digital good, outside of the segment(s). Fig. 7 illustrates two checkpoints CP1 and CP2 for the associated segments S1 and S2, respectively. Each checkpoint contains a function call to a verification function that, when executed, computes a checksum of the corresponding segment and compares that result with the precomputed checksum hidden in the digital good. The checkpoints therefore have knowledge of where the precomputed checksums are located. In practice, the precomputed checksums CS1 and CS2 may be located at the checkpoints, or separately from the checkpoints as illustrated.

When the client executes the digital good, the client-side evaluator 152 comes across the checkpoint and calls the verification function. If the checksums match, the digital good is assumed to be authentic; otherwise, the client is alerted that the digital good is not authentic and may be an illicit copy.

Acyclic (Dag-Based) Code Integrity Verification

Acyclic, or dag-based, code integrity verification is a tool that is rooted in the code integrity verification, but accommodates more complex nesting among the variously protected segments. "Dag" stands for "directed acyclic graph". Generally speaking, acyclic code integrity verification imposes an order to which the various checkpoints and checksum computations are performed to accommodate the complex nesting of protected segments.

Fig. 8 shows a portion of a digital good 800 having one segment S4 completely contained within another segment S3. The checkpoint CP4 for segment S4 is also contained within segment S3. In this nesting arrangement, executing checkpoint CP4 affects the bytes within the segment S3, which in turn

affects an eventual checksum operation performed by checkpoint CP3. Accordingly, evaluation of segment S3 is dependent on a previous evaluation of segment S4.

The acyclic code integrity verification tool 136(2) attempts to arrange the numerous evaluations in an order that handles all of the dependencies. The tool employs a topological sort to place the checkpoints in a linear order to ensure that dependencies are handled in an orderly fashion.

Cyclic Code Integrity Verification

Cyclic code-integrity verification extends dag-based verification by allowing cycles in the cross-verification graph. For example, if code segment S4 verifies code segment S5, and S5 also verifies S4, we have a cycle consisting of the nodes S4 and S5. With such cycles, a proper order for checksum computation does not exist. Thus, a topological sort does not suffice, and some checksums may be computed incorrectly. Cycles require an additional step to fix up any affected checksums.

One specific method of correcting checksums is to set aside and use some "free" space inside protected segments. This space, typically one or a few machine words, is part of the code bytes verified by checksum computation. If a particular checksum is incorrect, the extra words can be changed until the checksum becomes proper. While cryptographic hash functions are specifically designed to make this impractical, we can use certain cryptographic message authentication codes (MACs) as checksums to achieve this easily.

Secret Key Scattering

Secret key scattering is a tool that may be used to offer some security to a digital good. Cryptographic keys are often used by cryptography functions to code portions of a digital product. The tool scatters these cryptographic keys, in whole or in part, throughout the digital good in a manner that appears random and untraceable, but still allows the evaluator to recover the keys. For example, a scattered key might correspond to a short string used to compute indices into a pseudorandom array of bytes in the code section, to retrieve the bytes specified by the indices, and to combine these bytes into the actual key.

There are two types of secret key scattering methods: static and dynamic. Static key scattering methods place predefined keys throughout the digital good and associate those keys in some manner. One static key scattering technique is to link the scattered keys or secret data as a linked list, so that each key references a next key and a previous or beginning key. Another static key scattering technique is subset sum, where the secret key is converted into an encrypted secret data and a subset sum set containing a random sequence of bytes. Each byte in the secret data is referenced in the subset sum set. These static key scattering techniques are well known in the art.

Dynamic key scattering methods break the secret keys into multiple parts and then scatter those parts throughout the digital good. In this manner, the entire key is never computed or stored in full anywhere on the digital good. For instance, suppose that the digital good is encrypted using the well-known RSA public key scheme. RSA (an acronym for the founders of the algorithm) utilizes a pair of keys, including a public key e and a private key d. To encrypt and decrypt a message m, the RSA algorithm requires:

Encrypt: $y = m^e \ mod \ n$

Decrypt: $y^d = (m^e)^d \ mod \ n = m$

The secret key d is broken into many parts:

$$d = d_1 + d_2 + \ldots + d_k$$

The key parts $d_1, d_2, \ldots d_k$ are scattered throughout the digital good. To recover the message during decryption, the client computes:

$$y^d_1 = z_1$$
$$y^d_2 = z_2$$
$$\vdots$$
$$y^d_k = z_k$$

where, $m = z_1 \cdot z_2 \cdot \ldots z_k$

Obfuscated Function Execution

Another tool that may be used to protect a digital good is known as "obfuscated function execution". This tool subdivides a function into multiple blocks, which are separately encrypted by the secure processor. When executing the function, the secure processor uses multiple threads to decrypt each block into a random memory area while executing another block concurrently. More specifically, a first process thread decrypts the next block and temporarily stores

the decrypted block in memory. Simultaneously, a second process thread executes and then destroys the code in the current block.

The benefit of this tool is that only one block is visible at a time, while the other blocks remain encrypted. On the Intel x86 platform, code run in this manner should be self-relocatable, which means that function calls are typically replaced with calls via function pointers, or an additional program step fixes up any function calls that use relative addressing. Other platforms may have other requirements.

Code As An S-Box

Many ciphers (often block ciphers), including the Data Encryption Standard (DES), use one or more substitution boxes (S-boxes) to scramble data. The same S-box(es) are then also used to de-scramble the data. An S-box is essentially a table that maps *n*-bit binary strings onto a set of *m*-bit binary strings, where *m* and *n* are typically small integers. Depending on the cipher, S-boxes may be fixed or variable. Both S-boxes and code segments can be viewed simply as arrays of bytes, so an important code segment can be used as an S-box for a cipher to encrypt another important segment. If a cracker patches the segment serving as the S-box, the encrypted segment will be incorrectly decrypted. This is similar in spirit to using a segment's checksum as the decryption key for another segment, but is subtler and better obfuscated. In ciphers using multiple S-boxes, then other important segments (which may partially or wholly overlap one another) can each be used as an S-box in encrypting one or more other important segments.

In one implementation, an S-box is a table having multiple rows and multiple columns of values. The *n*-bit input to the S-box is used to identify a

particular row and column of the table, and the value stored at that row and column is the *m*-bit output of the S-box. For example, in DES S-boxes are often implemented with the first and last bits of the input (e.g., bits 1 and *n*) being used to form a 2-bit number identifying a row in the table, and the remaining bits (e.g., bits 2 through *n*-1) used to form a (*n*-2)-bit number identifying a column in the table. Alternatively, rows and columns can be identified in different manners.

The values stored in the table can be generated in a variety of different manners. By way of example, each byte (e.g., starting with the first byte) or other grouping of bits from the code segment being used to generate the S-box can be used as a value in the table. By way of another example, the number of bits used to generate each value in the table can be determined by identifying the total number of bits in the code segment being used to generate the S-box and dividing that sum by the number of table entries needed (e.g., and using the integer portion of the result as the number of bits to be used).

Any portion of a digital good can be used as an S-box. For example, an important portion of a video image, an important function of a program (e.g., a function that checks for the existence of a particular registration number, a function that outputs search results, etc.), etc. may be used. Which portions are determined to be important can vary based on the particular digital good.

The segment being encrypted can be part of the same digital good as the segment being used as the S-box, or alternatively can be part of another digital good. If a first segment being encrypted and a second segment being used as the S-box are part of the same digital good and both are to be encrypted, then care should be taken in selecting the first and/or second segments so that the second segment is de-scrambled (by use of another S-box) prior to de-scrambling the first

segment so that the correct values of the second segment can be used in subsequently de-scrambling the first segment. Alternatively, the scrambled values of the second segment could be used as the S-box for scrambling the first segment, in which case care should be taken to ensure that the first segment is de-scrambled prior to the second segment being de-scrambled.

Fig. 9 illustrates an exemplary implementation of using code as an S-box implemented by the code as an S-box tool 136(6) in the obfuscator 134. Initially, a first portion of the digital good is selected (block 902), and an S-box generated based on the values in the first portion (block 904). A second portion of the same (or another) digital good that is to be encrypted is then identified (block 906). The values of the second portion are then mapped to new "substitution" values based on the contents of the S-box (block 908).

Encryption/Decryption

Another tool to protect a digital good is encryption and decryption. This tool breaks the digital good into different chunks and then encrypts each chunk using different keys. The chunks might represent multi-layered and overlapping code sections. Checksums of code sections can serve as encryption keys.

Probabilistic Checking

The secure processor has its own pseudorandom-number generator (PRNG) that can be used to perform security actions, such as integrity verification, with certain probabilities. Probabilistic checking uses these probabilities to ensure that a protected program behaves differently during each run. For example, some checks could be during every run, others approximately every other run, and still

week or hour of the day. As the changes are made, the software product executes differently, even though it is performing essentially the same functions. Varying the execution path makes it difficult for an attacker to glean clues from repeatedly executing the product.

Anti-Debugging Methods

Anti-debugging methods are another tool that can be used to protect a digital good. Anti-debugging methods are very specific to particular implementations of the digital good, as well as the processor that the good is anticipated to run on.

As an example, the client-side secure processor may be configured to provide kernel-mode device drivers (e.g., a WDM driver for Windows NT and 2000, and a VxD for Windows 9x) that can redirect debugging-interrupt vectors and change the x86 processor's debug address registers. This redirection makes it difficult for attackers who use kernel debugging products, such as SoftICE. Additionally, the secure processor provides several system-specific methods of detecting Win32-API-based debuggers. Generic debugger-detection methods include integrity verification (to check for inserted breakpoints) and time analysis (to verify that execution takes an expected amount of time).

Separation in Time/Space of Tamper Detection and Response

Another tool that is effective for protecting digital goods is to separate the events of tamper detection and the eventual response. Separating detection and response makes it difficult for an attacker to discern what event or instruction set triggered the response.

These events may be separated in time, whereby tamper detection is detected at a first time and a response (e.g., halting execution of the product) is applied at some subsequent time. The events may also be separated in space, meaning that the detection and response are separated in the product itself.

Conclusion

Although the description above uses language that is specific to structural features and/or methodological acts, it is to be understood that the invention defined in the appended claims is not limited to the specific features or acts described. Rather, the specific features and acts are disclosed as exemplary forms of implementing the invention.

Various DRM techniques have been developed and employed in an attempt to thwart potential pirates from illegally copying or otherwise distributing the digital goods to others. For example, one DRM technique includes requiring the consumer to insert the original CD-ROM or DVD for verification prior to enabling the operation of a related copy of the digital good. Unfortunately, this DRM technique typically places an unwelcome burden on the honest consumer, especially those concerned with speed and productivity. Moreover, such techniques are impracticable for digital goods that are site licensed, such as software products that are licensed for use by several computers, and/or for digital goods that are downloaded directly to a computer. Additionally, it is not overly difficult for unscrupulous individuals/organizations to produce working pirated copies of the CD-ROM.

Another DRM technique includes requiring or otherwise encouraging the consumer to register the digital good with the provider, for example, either through the mail or online via the Internet or a direct connection. Thus, the digital good may require the consumer to enter a registration code before allowing the digital good to be fully operational or the digital content to be fully accessed. Unfortunately, such DRM techniques are not always effective since unscrupulous individuals/organizations need only break through or otherwise undermine the DRM protections in a single copy of the digital good. Once broken, copies of the digital good can be illegally distributed, hence such DRM techniques are considered to be Break-Once, Run-Everywhere (BORE) susceptible. Various different techniques can be used to defeat BORE, such as per-user software individualization, watermarks, etc. However, a malicious user may still be able to identify and remove from the digital good these various protections.

Accordingly, there remains a need for a technique that addresses the concerns of the publisher/developer, allowing alteration of the digital content to be identified to assist in protecting the content from many of the known and common attacks, but does not impose unnecessary and burdensome requirements on legitimate users.

SUMMARY

Using at least a portion of a digital good as a substitution box (S-box) is described herein.

According to one aspect, a portion of a digital good is selected to be used as a substitution box (S-box) in encrypting at least another portion of a digital good. The digital good being encrypted can be the same digital good, or alternatively a different digital good, than the digital good from which the portion used as an S-box is selected. During the encryption process, the S-box is used (often in the context of a block cipher) to substitute values of the portion being encrypted with new values (a process also referred to as "scrambling"). The bit pattern of the portion of the digital good being used as the S-box is used to determine, for each input value of the portion being encrypted (e.g., each byte), what substitute value should be used. Subsequently, when the digital good is being decrypted, if the portion of the digital good being used as the S-box has been modified (e.g., by a cracker trying to patch the portion), then the encrypted portion will not be de-scrambled properly and the decryption will fail.

BRIEF DESCRIPTION OF THE DRAWINGS

The same numbers are used throughout the drawings to reference like elements and features.

Fig. 1 is a block diagram of a DRM distribution architecture that protects digital goods by automatically and randomly obfuscating portions of the goods using various tools.

Fig. 2 is a block diagram of a system for producing a protected digital good from an original good.

Fig. 3 is a flow diagram of a protection process implemented by the system of Fig. 2.

Fig. 4 is a diagrammatical illustration of a digital good after being coded using the process of Fig. 3.

Fig. 5 is a diagrammatical illustration of a protected digital good that is shipped to a client, and shows an evaluation flow through the digital good that the client uses to evaluate the authenticity of the good.

Fig. 6 is a flow diagram of an oblivious checking process that may be employed by the system of Fig. 2.

Fig. 7 is a diagrammatic illustration of a digital good that is modified to support code integrity verification.

Fig. 8 is a diagrammatic illustration of a digital good that is modified to support cyclic code integrity verification.

Fig. 9 is a flow diagram of a process for using code as an S-box that may be employed by the system of Fig. 2.

others only occasionally. This makes the cracker's task much more difficult, since a program no longer exhibits definite, repeatable behavior between runs. In fact, a patched program may work properly once or twice, leading the cracker to believe that his efforts were successful; however, the program will fail in a subsequent run. This is part of an overall strategy of varying paths of execution between runs to complicate reverse engineering, as described elsewhere in this document. .

Boolean Check Obfuscation

Boolean checking utilizes Boolean functions to evaluate the authenticity of code sections or results generated from executing the code. A problem with Boolean checking is that an attacker can often identify the Boolean function and rewrite the code to avoid the Boolean check. According, the Boolean check obfuscation tool attempts to hide the Boolean function so that it is difficult to detect and even more difficult to remove.

Consider, for example, the following Boolean check that compares a register with a value "1" as a way to determine whether the digital good is authentic or a copy.

```
COMP reg1, 1
   BEQ good_guy
   (crash)

good_guy (go on)
```

In this example, if the compare operation is true (i.e., the Boolean check is valid), the program is to branch to "good_guy" and continue. If the compare is false, the program runs instructions that halt operation. To defeat this Boolean

check, an attacker merely has to change the "branch equal" or "BEQ" operation to a "branch always" condition, thereby always directing program flow around the "crash" instructions.

There are many ways to obfuscate a Boolean check. One approach is to add functions that manipulate the register values being used in the check. For instance, the following operations could be added to the above set of instructions:

```
SUB reg1, 1
ADD sp, reg1
     :
COMP reg1, 1
```

These instructions change the contents of register 1. If an attacker alters the program, there is a likelihood that such changes will disrupt what values are used to change the register contents, thereby causing the Boolean check to fail.

Another approach is to add "dummy" instructions to the code. Consider the following:

```
LEA reg2, good_guy
SUB reg2, reg1
INC reg2
JMP reg2
```

The "subtract", "increment", and "jump" instructions following the "load effective address" are dummy instructions that are essentially meaningless to the operation of the code.

A third approach is to employ jump tables, as follows:

```
MOV reg2, JMP_TAB[reg1]
JMP reg2
JMP_TAB: <bad_guy jump>
         <good_guy jump>
```

The above approaches are merely a few of the many different ways to obfuscate Boolean checks. Others may also be used.

In-Lining

The in-lining tool is useful to guard against single points of attack. The secure processor provides macros for inline integrity checks and pseudorandom generators. These macros essentially duplicate code, adding minor variations, which make it difficult to attack.

Reseeding of PRG With Time Varying Inputs

Many software products are designed to utilize random bit streams output by pseudo random number generators (PRGs). PRGs are seeded with a set of bits that are typically collected from multiple different sources, so that the seed itself approximates random behavior. One tool to make the software product more difficult to attack is to reseed the PRGs after every run with time varying inputs so that each pass has different PRG outputs.

Anti-Disassembly Methods

Disassembly is an attack methodology in which the attacker studies a print out of the software program and attempts to discover hidden protection schemes, such as code integrity verification, Boolean check obfuscation, and the like. Anti-

disassembly methods try to thwart a disassembly attack by manipulating the code is such a manner that it appears correct and legitimate, but in reality includes information that does not form part of the executed code.

One exemplary anti-disassembly method is to employ almost plaintext encryption that indiscreetly adds bits to the code (e.g., changing occasional opcodes). The added bits are difficult to detect, thereby making disassembly look plausible. However, the added disinformation renders the printout not entirely correct, rendering the disassembly practices inaccurate.

Another disassembly technique is to add random bytes into code segments and bypass them with jumps. This serves to confuse conventional straight-line disassemblers.

Shadowing

Another protection tool shadows relocatable addresses by adding "secret" constants. This serves to deflect attention away from crucial code sections, such as verification and encryption functions, that refer to address ranges within the executing code. Addition of constants (within a certain range) to relocatable words ensures that the loader still properly fixes up these words if an executable happens not to load at its preferred address. This particular technique is specific to the Intel x86 platform, but variants are applicable to other platforms.

Varying Execution Path Between Runs

One protection tool that may be employed to help thwart attackers is to alter the path of execution through the software product for different runs. As an example, the code may include operations that change depending on the day of